14 | 十面埋伏的并发：多线程真的很难吗？

2020-06-06 罗剑锋

讲述：Chrono
时长 15:09 大小 13.89M

你好，我是 Chrono。

今天，我们来聊聊“并发”（Concurrency）、“多线程”（multithreading）。

在 20 年前，大多数人（当然也包括我）对这两个词还是十分陌生的。那个时候，CPU 的性能不高，要做的事情也比较少，没什么并发的需求，简单的单进程、单线程就能够解决大多数问题。

但到了现在，计算机硬件飞速发展，不仅主频上 G，还有了多核心，运算能力大幅度提升，只使用单线程很难“喂饱”CPU。而且，随着互联网、大数据、音频视频处理等新需求的不断涌现，运算量也越来越大。这些软硬件上的变化迫使“并发”“多线程”成为了每个技术人都不得不面对的课题。

通俗地说，“并发”是指在一个时间段里有多个操作在同时进行，与“多线程”并不是一回事。

并发有很多种实现方式，而多线程只是其中最常用的一种手段。不过，因为多线程已经有了很多年的实际应用，也有很多研究成果、应用模式和成熟的软硬件支持，所以，对这两者的区分一般也不太严格，下面我主要来谈多线程。

认识线程和多线程

要掌握多线程，就要先了解线程（thread）。

线程的概念可以分成好几个层次，从 CPU、操作系统等不同的角度看，它的定义也不同。今天，我们单从语言的角度来看线程。

在 C++ 语言里，线程就是一个能够独立运行的函数。比如你写一个 lambda 表达式，就可以让它在线程里跑起来：

复制代码

```
auto f = []()                  // 定义一个lambda表达式
{
    cout << "tid=" <<
        this_thread::get_id() << endl;
};

thread t(f);                   // 启动一个线程，运行函数f
```

任何程序一开始就有一个主线程，它从 main() 开始运行。主线程可以调用接口函数，创建出子线程。子线程会立即脱离主线程的控制流程，单独运行，但共享主线程的数据。程序创建出多个子线程，执行多个不同的函数，也就成了多线程。

多线程的好处你肯定能列出好几条，比如任务并行、避免 I/O 阻塞、充分利用 CPU、提高用户界面响应速度，等等。

不过，多线程也对程序员的思维、能力提出了极大的挑战。不夸张地说，它带来的麻烦可能要比好处更多。

这个问题相信你也很清楚，随手就能数出几个来，比如同步、死锁、数据竞争、系统调度开销等……每个写过实际多线程应用的人，可能都有“一肚子的苦水”。

其实，多线程编程这件事“说难也不难，说不难也难”。这句话听上去好像有点自相矛盾，但却有一定的道理。为什么这么说呢？

说它不难，是因为线程本身的概念是很简单的，只要规划好要做的工作，不与外部有过多的竞争读写，很容易就能避开“坑”，充分利用多线程，“跑满”CPU。

说它难，则是因为现实的业务往往非常复杂，很难做到完美的解耦。一旦线程之间有共享数据的需求，麻烦就接踵而至，因为要考虑各种情况、用各种手段去同步数据。随着线程数量的增加，复杂程度会以几何量级攀升，一不小心就可能会导致灾难性的后果。

多线程涵盖的知识点太多，许多大师、高手都不敢自称精通，想用一节课把多线程开发说清楚是完全不可能的。

所以，今天我们只聚焦 C++ 的标准库，了解下标准库为多线程编程提供了哪些工具，在语言层面怎么改善多线程应用。有了这个基础，你再去看那些专著时，就可以省很多力气，开发时也能少走些弯路。

首先，你要知道一个最基本但也最容易被忽视的常识：**“读而不写”就不会有数据竞争**。

所以，在 C++ 多线程编程里读取 const 变量总是安全的，对类调用 const 成员函数、对容器调用只读算法也总是线程安全的。

知道了这一点，你就应该多实践第 7 讲里的做法，多用 const 关键字，尽可能让操作都是只读的，为多线程打造一个坚实的基础。

然后，我要说一个多线程开发的原则，也是一句“自相矛盾”的话：

> 最好的并发就是没有并发，最好的多线程就是没有线程。

这又是什么意思呢？

简单来说，就是在大的、宏观的层面上“看得到”并发和线程，而在小的、微观的层面上“看不到”线程，减少死锁、同步等恶性问题的出现几率。

多线程开发实践

下面，我就来讲讲具体该怎么实践这个原则。在 C++ 里，有四个基本的工具：仅调用一次、线程局部存储、原子变量和线程对象。

仅调用一次

程序免不了要初始化数据，这在多线程里却是一个不大不小的麻烦。因为线程并发，如果没有某种同步手段来控制，会导致初始化函数多次运行。

为此，C++ 提供了“仅调用一次”的功能，可以很轻松地解决这个问题。

这个功能用起来很简单，你要先声明一个 **once_flag** 类型的变量，最好是静态、全局的（线程可见），作为初始化的标志：

复制代码

```
static std::once_flag flag;         // 全局的初始化标志
```

然后调用专门的 **call_once()** 函数，以函数式编程的方式，传递这个标志和初始化函数。这样 C++ 就会保证，即使多个线程重入 call_once()，也只能有一个线程会成功运行初始化。

下面是一个简单的示例，使用了 lambda 表达式来模拟实际的线程函数。你可以把 GitHub 仓库里的代码下到本地，实际编译运行看看效果：

复制代码

```
auto f = []()                   // 在线程里运行的lambda表达式
{
    std::call_once(flag,        // 仅一次调用，注意要传flag
        [](){                   // 匿名lambda，初始化函数，只会执行一次
            cout << "only once" << endl;
        }                       // 匿名lambda结束
    );                          // 在线程里运行的lambda表达式结束
};

```

```
thread t1(f);            // 启动两个线程，运行函数f
thread t2(f);
```

call_once() 完全消除了初始化时的并发冲突，在它的调用位置根本看不到并发和线程。所以，按照刚才说的基本原则，它是一个很好的多线程工具。

它也可以很轻松地解决多线程领域里令人头疼的“双重检查锁定”问题，你可以自己试一试，用它替代锁定来初始化。

线程局部存储

读写全局（或者局部静态）变量是另一个比较常见的数据竞争场景，因为共享数据，多线程操作时就有可能导致状态不一致。

但如果仔细分析的话，你会发现，有的时候，全局变量并不一定是必须共享的，可能仅仅是为了方便线程传入传出数据，或者是本地 cache，而不是为了共享所有权。

换句话说，这应该是线程独占所有权，不应该在多线程之间共同拥有，术语叫“**线程局部存储**”（thread local storage）。

这个功能在 C++ 里由关键字 **thread_local** 实现，它是一个和 static、extern 同级的变量存储说明，有 thread_local 标记的变量在每个线程里都会有一个独立的副本，是“线程独占”的，所以就不会有竞争读写的问题。

下面是示范 thread_local 的代码，先定义了一个线程独占变量，然后用 lambda 表达式捕获引用，再放进多个线程里运行：

复制代码

```
thread_local int n = 0;          // 线程局部存储变量

auto f = [&](int x)              // 在线程里运行的lambda表达式，捕获引用
{
    n += x;                      // 使用线程局部变量，互不影响
    cout << n;                   // 输出，验证结果
};

thread t1(f, 10);              // 启动两个线程，运行函数f
thread t2(f, 20);
```

在程序执行后，我们可以看到，两个线程分别输出了 10 和 20，互不干扰。

你可以试着把变量的声明改成 static，再运行一下。这时，因为两个线程共享变量，所以 n 就被连加了两次，最后的结果就是 30。

```
static int n = 0;     // 静态全局变量
...                    // 代码与刚才的相同
```

和 call_once() 一样，thread_local 也很容易使用。但它的应用场合不是那么显而易见的，这要求你对线程的共享数据有清楚的认识，区分出独占的那部分，消除多线程对变量的并发访问。

原子变量

那么，对于那些非独占、必须共享的数据，该怎么办呢？

要想保证多线程读写共享数据的一致性，关键是**要解决同步问题**，不能让两个线程同时写，也就是“互斥”。

这在多线程编程里早就有解决方案了，就是互斥量（Mutex）。但它的成本太高，所以，对于小数据，应该采用“**原子化**”这个更好的方案。

所谓原子（atomic），在多线程领域里的意思就是不可分的。操作要么完成，要么未完成，不能被任何外部操作打断，总是有一个确定的、完整的状态。所以也就不会存在竞争读写的问题，不需要使用互斥量来同步，成本也就更低。

但不是所有的操作都可以原子化的，否则多线程编程就太轻松了。目前，C++ 只能让一些最基本的类型原子化，比如 atomic_int、atomic_long，等等：

```
using atomic_bool = std::atomic<bool>;     // 原子化的bool
using atomic_int  = std::atomic<int>;       // 原子化的int
using atomic_long = std::atomic<long>;     // 原子化的long
```

这些原子变量都是模板类 atomic 的特化形式，包装了原始的类型，具有相同的接口，用起来和 bool、int 几乎一模一样，但却是原子化的，多线程读写不会出错。

注意，我说了“几乎”这个词。它还是有些不同的，一个重要的区别是，**原子变量禁用了拷贝构造函数，所以在初始化的时候不能用“=”的赋值形式，只能用圆括号或者花括号**：

```
atomic_int  x {0};          // 初始化，不能用=
atomic_long y {1000L};      // 初始化，只能用圆括号或者花括号

assert(++x == 1);           // 自增运算

y += 200;                   // 加法运算
assert(y < 2000);           // 比较运算
```

除了模拟整数运算，原子变量还有一些特殊的原子操作，比如 store、load、fetch_add、fetch_sub、exchange、compare_exchange_weak/compare_exchange_strong，最后一组就是著名的 CAS（Compare And Swap）操作。

而另一个同样著名的 TAS（Test And Set）操作，则需要用到一个特殊的原子类型 atomic_flag。

它不是简单的 bool 特化（atomic），没有 store、load 的操作，只用来实现 TAS，保证绝对无锁。

你能用这些原子变量做些什么呢？

最基本的用法是把原子变量当作线程安全的全局计数器或者标志位，这也算是“初心”吧。但它还有一个更重要的应用领域，就是实现高效的无锁数据结构（lock-free）。

但我**强烈不建议**你自己尝试去写无锁数据结构，因为无锁编程的难度比使用互斥量更高，可能会掉到各种难以察觉的“坑”（例如 ABA）里，最好还是用现成的库。

遗憾的是，标准库在这方面帮不了你，虽然网上可以找到不少开源的无锁数据结构，但经过实际检验的不多，我个人觉得你可以考虑 **boost.lock_free**。

线程

到现在我说了 call_once、thread_local 和 atomic 这三个 C++ 里的工具，它们都不与线程直接相关，但却能够用于多线程编程，尽量消除显式地使用线程。

但是，必须要用线程的时候，我们也不能逃避。

C++ 标准库里有专门的线程类 thread，使用它就可以简单地创建线程，在名字空间 std::this_thread 里，还有 yield()、get_id()、sleep_for()、sleep_until() 等几个方便的管理函数。因为它们的用法比较简单，资料也随处可见，我就不再重复了。

下面的代码同时示范了 thread 和 atomic 的用法：

复制代码

```
static atomic_flag flag {false};    // 原子化的标志量
static atomic_int  n;               // 原子化的int

auto f = [&]()              // 在线程里运行的lambda表达式，捕获引用
{
    auto value = flag.test_and_set();  // TAS检查原子标志量

    if (value) {
        cout << "flag has been set." << endl;
    } else {
        cout << "set flag by " <<
            this_thread::get_id() << endl;  // 输出线程id
    }

    n += 100;                   // 原子变量加法运算

    this_thread::sleep_for(     // 线程睡眠
        n.load() * 10ms);       // 使用时间字面量
    cout << n << endl;
};                          // 在线程里运行的lambda表达式结束

thread t1(f);               // 启动两个线程，运行函数f
thread t2(f);

t1.join();                  // 等待线程结束
t2.join();
```

但还是基于那个原则，我建议你不要直接使用 thread 这个“原始”的线程概念，最好把它隐藏到底层，因为“看不到的线程才是好线程”。

具体的做法是调用函数 async()，它的含义是“**异步运行**”一个任务，隐含的动作是启动一个线程去执行，但不绝对保证立即启动（也可以在第一个参数传递 std::launch::async，要求立即启动线程）。

大多数 thread 能做的事情也可以用 async() 来实现，但不会看到明显的线程：

复制代码

```
auto task = [](auto x)                    // 在线程里运行的lambda表达式
{
    this_thread::sleep_for( x * 1ms);  // 线程睡眠
    cout << "sleep for " << x << endl;
    return x;
};

auto f = std::async(task, 10);          // 启动一个异步任务
f.wait();                               // 等待任务完成

assert(f.valid());                     // 确实已经完成了任务
cout << f.get() << endl;               // 获取任务的执行结果
```

其实，这还是函数式编程的思路，在更高的抽象级别上去看待问题，异步并发多个任务，让底层去自动管理线程，要比我们自己手动控制更好（比如内部使用线程池或者其他机制）。

async() 会返回一个 future 变量，可以认为是代表了执行结果的“期货”，如果任务有返回值，就可以用成员函数 get() 获取。

不过要特别注意，get() 只能调一次，再次获取结果会发生错误，抛出异常 std::future_error。（至于为什么这么设计我也不太清楚，没找到官方的解释）

另外，这里还有一个很隐蔽的“坑”，如果你不显式获取 async() 的返回值（即 future 对象），它就会**同步阻塞**直至任务完成（由于临时对象的析构函数），于是“async”就变成了“sync”。

所以，即使我们不关心返回值，也总要用 auto 来配合 async()，避免同步阻塞，就像下面的示例代码那样：

复制代码

```
std::async(task, ...);              // 没有显式获取future，被同步阻塞
```

```
2  auto f = std::async(task      );    // 只有上一个任务完成后才能被执行
```

标准库里还有 mutex、lock_guard、condition_variable、promise 等很多工具，不过它们大多数都是广为人知的概念在 C++ 里的具体实现，用法上没太多新意，所以我就不再多介绍了。

小结

说了这么长时间，你可能会有些奇怪，这节课的标题里有线程，但我并没有讲太多线程相关的东西，更多的是在讲“不用线程”的思维方式。

所谓“当局者迷”，如果你一头扎进多线程的世界，全力去研究线程、互斥量、锁等细节，就很容易“钻进死胡同”“一条道走到黑”。

很多时候，我们应该跳出具体的编码，换个角度来看问题，也许就能够“柳暗花明又一村”，得到新的、优雅的解决办法。

好了，今天就到这里，做个小结：

1. 多线程是并发最常用的实现方式，好处是任务并行、避免阻塞，坏处是开发难度高，有数据竞争、死锁等很多“坑”；
2. call_once() 实现了仅调用一次的功能，避免多线程初始化时的冲突；
3. thread_local 实现了线程局部存储，让每个线程都独立访问数据，互不干扰；
4. atomic 实现了原子化变量，可以用作线程安全的计数器，也可以实现无锁数据结构；
5. async() 启动一个异步任务，相当于开了一个线程，但内部通常会有优化，比直接使用线程更好。

我再告诉你一个好消息：C++20 正式加入了协程（关键字 co_wait/co_yield/co_return）。它是用户态的线程，没有系统级线程那么多的麻烦事，使用它就可以写出开销更低、性能更高的并发程序。让我们一起期待吧！

课下作业

最后是课下作业时间，给你留两个思考题：

1. 你在多线程编程的时候遇到过哪些“坑”，有什么经验教训?

2. 你觉得 async() 比直接用 thread 启动线程好在哪里?

欢迎在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎分享给你的朋友。我们下节课见。

课外小贴士

1. 在C语言（C11，不是ANSI C）里也有“thread_local”的功能，但它只是一个宏，GCC也有自己的“__thread”关键字。

2. 原子一词来自于希腊语，意思是不可分的，虽然近代物理学早已经证明原子可以进一步分成质子、中子、电子，但这个词的意思却被沿用到了其他的领域。

3. C++11里有一个专门用来初始化原子变量的函数atomic_init()，但在C++20里，它已经被废弃。

4. 如果只是想简单地在线程里启动一个异步任务，完全不关心返回值，可以调用thread的成员函数detach()，比async()会方便一点。

5. tbb（Threading Building Blocks）是Intel开发的一个多线程库，提供线程安全的容器、并行算法等一些有用的工具。

极客时间

更多课程推荐

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　13 | 五花八门的算法：不要再手写for循环了

下一篇　15 | 序列化：简单通用的数据交换格式有哪些？

精选留言 (11)

写留言

秦奋
2020-06-06

子线程会立即脱离主线程的控制流程，单独运行，但共享主线程的数据。这里指的是全局变量吧，像局部变量应该不行吧？

展开

作者回复: 也不一定，看传给子线程什么参数了，局部变量也可以给子线程共享。

4 2

被讨厌的勇气

2020-06-06

当需要获取线程的结果时，使用async可以直接获取其结果；使用thread则需要通过共享数据来获取，需要使用锁、条件变量。

async的缺点是只能获取一次。若需要保证线程一直运行，多次获取其‘结果’时，只能使用thread + condition_variable,不知道这样理解对不对？...

展开

作者回复:

1.基本正确，async()是一种相当简化的线程用法，目的就是获取future值，如果不是这个场景就不如thread灵活。

2.比如说线程里要保存一些cache数据，很显然这些cache不会是多线程共享的，用thread_local就比较好。可以把它理解成是专门给线程准备的static全局变量。

1

禾桃

2020-06-08

#1

请conditional_variable的

notify,wait这两个成员函数，在哪些情况下会引起context switch?

#2...

展开

作者回复:

1.问题太高端，不太理解context switch是什么，抱歉。

2.原子变量往细了说也是很复杂的，有好几种内存访问顺序规定，如果是最严格的memory_order_seq_cst就是你说的那样了。

2

泡泡龙
2020-06-07

虽然没有C++的多线程经验，但是记得其他语言的多线程，用IDE打断点调试是一个极难的问题

作者回复: 多线程调试从来都不是个简单的工作，一般都是打日志，里面列出线程号和使用的变量，然后再慢慢分析。

1coma
2020-06-07

罗老师，请问：
1.针对ABBA锁的问题，try-lock的方式一定比消除ABBA锁差吗?
2.针对多线程服务卡顿（高并发下）的问题，除了通过日志排查，有没有什么技巧?
展开

作者回复:
1.这个我也说不太好，我一般都用try-lock，感觉比较安全，不容易犯错。

2.可以用性能分析工具，比如perf、systemtap，后面会讲。

青生先森
2020-06-07

老师你好，假如一个变量用volatile修饰，是不是就可以不用加锁?

作者回复: 这种想法是非常危险的，一定要特别注意。

volatile只是表示变量易变，与线程没有任何关系。

不加锁应该用atomic变量。

有学识的兔子
2020-06-06

1. 我遇到一个接口内部使用了同步锁的同时，内部嵌套的接口再一次使用了该同步锁；会出现死锁；
2.除了隐藏细节外，方便实现和处理异步任务？

展开

作者回复:
1.互斥锁用起来真的是有很多要注意的地方，并发编程对程序员的思维是一大挑战。
这里我觉得可以用递归锁，就是可以重复加锁的锁，可以查一下标准文档。

2.我觉得async相当于是一个外观模式，屏蔽了底层细节，更容易使用，即使你不了解多线程，也能够利用线程开发出并发程序。

lckfa李钊
2020-06-06

线程 好开，不好关，Windows下有直接回收线程句柄强行关闭线程的api，但是很明显，这不是什么好的实践。线程最好的关闭方式 是让它自然消亡。
对于某些场景我们需要使用守护线程或者监控线程，里面跑着要一直循环执行的任务，那么这个任务怎么停止就是个大问题。假设我们有一个ui线程，里面有些指针，而这些指针被守护线程里的无限循环任务使用，当UI线程被关闭，这些指针被UI线程回收，而守护...

展开

作者回复:
1.thread的概念已经被大多数人熟知了，而async()用的比较少，所以我想让大家多了解它，让它在合适的地方去替代thread，简化多线程编程。

当然了，aysnc()也是不可能完全替代thread，各有优缺点吧。

2.node.js里面用的是epoll，跟线程没啥关系，等到C++20、23把networking和协程弄完善了，我们也可以用C++轻松写出高效的并发代码。

青生先森
2020-06-06

启动线程中的运行函数，普通函数，静态函数，成员函数有什么区别吗

作者回复: 没有本质区别，只是传入的参数不同。

TC128
2020-06-06

老师好，请问call_once里面的函数调用失败（出错）了咋办？只能重启软件吗？

作者回复: 看自己的业务逻辑吧，可以重试，或者报错、记日志。

刘丹
2020-06-06

能否介绍一下C++里的反射？

展开

作者回复: C++11/14里还没有这个特性，最新的C++20好像也没有，抱歉没办法帮到你。

1